Today's Topics

1. **Decision-Making**: IF, IF Else, If Elif Else
2. **Repeating Actions** : While Loop and For Loop

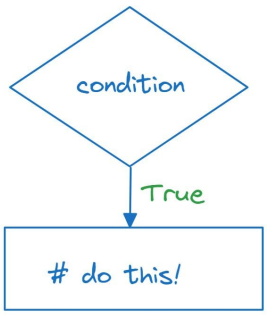
Decision Structures

Decision Structures are used to control the flow of execution based on certain conditions. They allow programs to make decisions and execute different blocks of code depending on whether a condition is `True` or `False`.

1. **`if` statement**
2. **`if...else` statement**
3. **`if..elif...else` Statement**
4. **nested `if` statement**

if statement

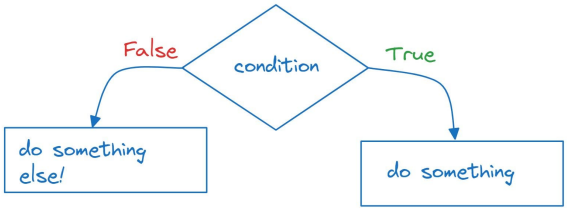
If the condition (Boolean expressions) is true, statement is executed; otherwise it is **skipped**




```
#This program illustrates if statement.
'''
If condition:
    # code to execute if condition is True
'''
number = int(input("Enter a number:"))
if number < 0:
    number =  - number
    print("The absolute value of number is", number)

    age = 18
if age >= 18:
    print("You are eligible to vote.")
```

if-else Statement

The condition is evaluated first. If the condition is true, statement1 is executed. If the condition is false, statement 2 is executed. Used to execute one block of code if the condition is `True`, and another if it is `False`

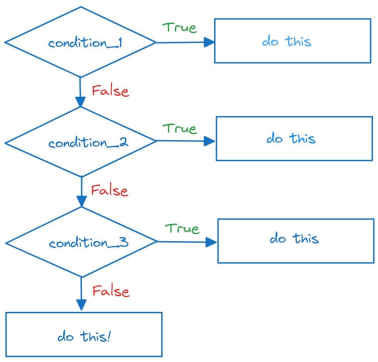



```
'''
if condition:
    # code if True
else:
    # code if False
'''
# This program test a number is even or odd.
number = int(input('Enter number: '))

if number%2 == 0:
    print(number, 'is even.')
else:
    print(number, 'is odd.')

age = 16
if age >= 18:
    print("You can vote.")
else:
```

```
    print("You cannot vote yet.")
```

if-elif-else Statement

The if-elif-else statement is used when you have **multiple conditions** to check. It allows you to test multiple conditions and execute the corresponding block of code for the first true condition encountered. If the conditions in the if and all elif statements evaluate to false, the `else` block is executed. **elif** is an abbreviation of "else if." Again, exactly one branch will be executed. There is no limit of the number of elif statements, but the last branch has to be an else statement. Used when you have **multiple conditions** to check



```
'''
if condition1:
    # code
elif condition2:
    # code
else:
    # code
'''
# This program determines greater number.
x = int(input("Enter first number:"))
y = int(input("Enter second number:"))

if x < y:
    print (x, "is less than", y)
elif x > y:
    print (x, "is greater than", y)
else:
    print (x, "and", y, "are equal")

marks = 75
if marks >= 90:
    print("Grade: A")
elif marks >= 80:
    print("Grade: B")
elif marks >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

Nested Decision Structures

One **conditional** can also be nested. Nested if statements are used when you need to check **multiple conditions**

within another condition. An `if` statement inside another `if` statement.

```
# This program determines greater number.
x = int(input("Enter first number:"))
y = int(input("Enter second number:"))

if x == y:
    print (x, "and", y, "are equal")
else:
    if x < y:
        print (x, "is less than", y)
    else:
        print (x, "is greater than", y)
```

Logical Operators in Condition Tests

Logical operators are often used in **condition tests** (especially in `if`, `elif`, and `else` statements) to combine or modify conditions

"and" Operator

It returns True if **all the conditions are true**. Otherwise, it returns False

```
# Checking Age and Salary for Loan Eligibility
age = 25
salary = 50000

if age >= 18 and salary >= 30000:
    print("You are eligible for a loan.")
```

```
else:
    print("You are not eligible for a loan.")

age = 25

if age >= 18 and age <= 60:
    print("You are eligible to work.")
```

"or" Operator

It returns True if **at least one** of the conditions is true, and False if all the conditions are false

```
# Checking if a Student has Passed in at Least One Sub
subject1 = 70
subject2 = 55

if subject1 >= 50 or subject2 >= 50:
    print("The student has passed in at least one subj
else:
    print("The student has failed in all subjects.")

day = "Sunday"

if day == "Saturday" or day == "Sunday":
    print("It's a weekend!")
```

"not" Operator

It returns True if the condition is false, and False if the condition is true.

```
answer = 17
if answer != 42:
    print("That is not the correct answer. Please try

is_raining = True

if not is_raining:
    print("You don't need an umbrella.")
else:
    print("Take an umbrella.")
```

Repeating Actions

Like running a block of code multiple times — you use **looping structures**. These are essential for automating repetitive tasks, such as.

1. Processing items in a list.
2. Repeating until a condition is met
3. Running a block of code a fixed number of times

We often require a set of statements to be executed **a number of times**. This type of program execution is called looping. Python provides the following construct.

Types of Loops in Python

1. **`for` loop** : for iterating over a sequence (like lists, strings, ranges, etc.).
2. **`while` loop** : for repeating as long as a condition is `True`

Also, **loop control statements** like `break`, `continue`, and `pass` help manage loop behavior.

The while Loop

Use when you want to **repeat until a condition is no longer true** . If we want to execute a piece of code as long as **the condition is true**, we can use a while loop. There are three different kinds of loops

1. **Count-controlled loops** : repeating a loop for a certain number of times.
2. **Condition-controlled loop** : repeated until a given condition changes
3. **Collection-controlled loop** : through the elements of a 'collection', which can be an array, list or other ordered sequence

```
'''
while condition:
    # code to repeat
'''

# This program print message 5 times.
i = 1
while i <= 5:
    print("I love programming in Python!")
    i = i + 1
```

Be careful with **infinite loops** — always ensure the condition will eventually become `False`.

```
# This program print n natural numbers.
n = int(input('Enter the value of n: '))
i = 1
while i <= n:
    print(i, end=' ')
    i = i + 1
```

```
# # Using a while loop with an existing iterable
student_names = ["Alice", "Bob", "Charlie", "David"]
index = 0

## the use of `len()` function to get the length of th
while index < len(student_names):
    print("Student:", student_names[index])
    index += 1
```

```
# another
# The list method `pop` removes and returns the last e
while student_names:
    current_student = student_names.pop()
    print("Current Student:", current_student)

print("All students have been processed.")
```

For Loop

Use when you **know how many times** you want to repeat an action. A for loop is used for iterating **over a sequence** (such as a list, tuple, dictionary, set, or string). First, we can generate a sequence using **range() function**. The general format range (start, stop, step) takes three arguments. Start and Step are the optional arguments. A start argument is a starting number of the sequence. If start is not specified it starts with 0. The stop argument is ending number of sequence. The step argument is linear difference of numbers. The default value of the step is 1 if not specified

Range(start, stop, step)

```
'''
```

```
for variable in sequence:
    # code to repeat
'''

for i in range (10):
     print(i, end = ' ')

for i in range(2, 10):
    print(i, end = ' ')

for i in range(3, 31, 3):
    print(i, end = ' ')

# This program prints backward counting
for i in range(5,0,-1):
    print(i)

# Loop through a string
for char in "Python":
    print(char)
```

Loop through List, Tuple, Dictionary

```
student_names = ["Alice", "Bob", "Charlie", "David"]

for name in student_names:
    print("Student:", name)
```

Nested Loops

A loop that inside another loop is called a nested loop.

```
# This program prints pattern
for i in range(1,4):
    for j in range(1,5):
        print(j, end = ' ')
    print()
```

Create a result from for loop

```
result = []
for k in range(2, 11):
    result.append(k)

print(result)
```

Loop Intervention

1. **`break`**: Stop the loop early.
2. **`continue`**: Skip the current iteration

break exits the loop prematurely, and **continue** skips the rest of the current iteration and moves to the next one.

```
student_names = ["Alice", "Bob", "Charlie", "David"]

for name in student_names:
    if name == "Charlie":
        break
    print(name)

for i in range(10):
    if i == 5:
        break
    print(i)

for number in range(1, 20):
if number == 12:
    continue
    print(number)
# This loop skips the number 12 and continues printing
```

```
for i in range(5):
    if i == 2:
        continue
    print(i)

# Using **continue** in a Loop
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue

print(current_number)
```

Nest Loops (Mathematical Operations)

A loop inside another loop.

```
for i in range(1, 4):
    for j in range(1, 3):
        print(f"i = {i}, j = {j}")
```

Do-While Loop Behavior

There is no built-in **do-while** loop like in some other programming languages. However, you can achieve the same behavior using a while loop with a break statement. Here's how you can emulate a **do-while loop in Python.**

Executes the block of code at least once

```
'''
while True:
    # code to execute at least once

    if not condition:
        break
'''
while True:
    user_input = input("Enter 'exit' to stop: ")
    if user_input == 'exit':
```

```
        break

while True:
    number = int(input("Enter a positive number: "))
    if number > 0:
        print("Thank you!")
        break
    else:
        print("That's not positive. Try again.")
```

This approach is similar to a do-while loop where the loop body is guaranteed to execute at least once before the condition is checked